Testing Smart Contracts: Which Technique Performs Best?

Citation for published version:
Akca, S, Peng, C & Rajan, A 2021, Testing Smart Contracts: Which Technique Performs Best? in *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM).*, 21, ESEM '21, Association for Computing Machinery, Inc, New York, NY, USA, 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, 2021, 11/10/21. https://doi.org/10.1145/3475716.3475779

Digital Object Identifier (DOI):
10.1145/3475716.3475779

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)

Testing Smart Contracts: Which Technique Performs Best?

Sefa Akca
University of Edinburgh
Edinburgh, United Kingdom
s.akca@sms.ed.ac.uk

Chao Peng
University of Edinburgh
Edinburgh, United Kingdom
chao.peng@ed.ac.uk

Ajitha Rajan
University of Edinburgh
Edinburgh, United Kingdom
arajan@ed.ac.uk

ABSTRACT

Background: Executing, verifying and enforcing credible transactions on permissionless blockchains is done using smart contracts. A key challenge with smart contracts is ensuring their correctness and security. Several test input generation techniques for detecting vulnerabilities in smart contracts have been proposed in the last few years. However, a comparison of proposed techniques to gauge their effectiveness is missing. **Aim:** This paper conducts an empirical evaluation of testing techniques for smart contracts. The testing techniques we evaluated are: (1) Blackbox fuzzing, (2) Adaptive fuzzing, (3) Coverage-guided fuzzing with an SMT solver and (4) Genetic algorithm. We do not consider static analysis tools, as several recent studies have assessed and compared effectiveness of these tools. **Method:** We evaluate effectiveness of the test generation techniques using (1) Coverage achieved - we use four code coverage metrics targeting smart contracts, (2) Fault finding ability - using artificially seeded and real security vulnerabilities of different types. We used two datasets in our evaluation - one with 1665 real smart contracts from Etherscan, and another with 90 real contracts with known vulnerabilities to assess fault finding ability. **Result:** We find Adaptive fuzzing performs best in terms of coverage and fault finding over contracts in both datasets. **Conclusion:** However, we believe considering dependencies between functions and handling Solidity specific features will help improve the performance of all techniques considerably.

CCS CONCEPTS

• **Security and privacy** → **Software security engineering**.

KEYWORDS

Blockchain, Ethereum, Smart Contract, Genetic Algorithm, Fault Seeding, Input Generation, Fuzzer, Constraint Solver

ACM Reference Format:
Sefa Akca, Chao Peng, and Ajitha Rajan. 2021. Testing Smart Contracts: Which Technique Performs Best?. In *ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM) (ESEM '21), October 11–15, 2021, Bari, Italy.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3475716.3475779

ESEM '21, October 11–15, 2021, Bari, Italy
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8665-4/21/10…$15.00
https://doi.org/10.1145/3475716.3475779

1 INTRODUCTION

A blockchain is a distributed ledger that stores a growing list of unmodifiable records called blocks that are linked to previous blocks. Executing, verifying and enforcing credible transactions on blockchains is done using smart contracts[1] [44].

A key challenge in developing contracts is to ensure that they are correct and free of security vulnerabilities, as bugs in their implementation may result in substantial financial losses. However, their security and trustworthiness is still in question. For instance, failure of the contract, DAO [42], due to unsafe design choices resulted in losses of, approximately, $50 million. Many other vulnerabilities in contracts have been reported recently [6, 11], like the Fomo3D attack in 2018 that led to a loss of $4 million. Contracts can handle large numbers of virtual coins, which provides enough financial incentive for attacks.

Several smart contract test input generation techniques have been proposed in the literature to detect security vulnerabilities [2, 13, 25, 34]. Many of the existing techniques use fuzzing approaches, namely (1) Blackbox Fuzzing (BF) - that relies on random input generation based on contract interface [2, 13, 25], and (2) Adaptive Fuzzing (AF) - feedback-guided input generation to generate high quality inputs that combines genetic algorithms with fuzzing [34]. Coverage-guided fuzzing and genetic algorithms have been used effectively in other domains to generate high quality inputs [1, 19, 20, 29, 35, 40]. Consequently, we decided to support these additional approaches for testing smart contracts. We implemented the following techniques in this paper - (3) Coverage-Guided Fuzzing with the aid of an SMT solver (GF), (4) Genetic algorithm (GA) using opcode coverage as an objective.

In this paper, we aim to compare the above four input generation approaches for Solidity[2] contracts with respect to their effectiveness in achieving code coverage, fault finding and overhead incurred. All the input generation approaches mentioned, except BF, are associated with a single implementation for Solidity - sFuzz [34] for AF, our native implementations for each of GF and GA. There are three available implementations/tools for Solidity testing using BF. For ease of representation and comparison, we pick a single BF implementation to use in our experiments – SolAnalyser [2]. SolAnalyser is an automated BF tool that generates inputs for all transactions and functions in Solidity contracts. We picked this tool over other BF tools [13, 25] as it achieved the best average fault finding over the contracts in our dataset and was easy to use with a well maintained implementation.

We evaluated the four input generation techniques using two datasets of Solidity contracts – (1) Random-C comprising 1665 Solidity contracts with no known vulnerabilities, (2) Vulnerable-C containing 90 Solidity contracts with known vulnerabilities. We

[1] Smart contracts are programs that execute on the Ethereum network.
[2] Solidity is a popular high-level programming language for writing Ethereum smart contracts.

generated inputs with each of the techniques for a fixed time of 15 seconds per contract. We measured four Solidity code coverage metrics - branch coverage, opcode coverage, call coverage, event coverage achieved by the inputs over contracts in both datasets. We also seeded artificial faults for different types of security vulnerabilities in the Random-C dataset. We then assessed the effectiveness of the input generation techniques in uncovering the seeded faults in Random-C contracts. To mitigate the problem of artificial faults not being representative of real faults, we additionally performed fault finding on the Vulnerable-C dataset of 90 real contracts with known vulnerabilities gathered by authors in [16]. Among the four input generation approaches, we found AF was most effective at contract coverage and fault finding over both the Random-C and Vulnerable-C datasets. AF also generated the smallest input set with least overhead in execution.

2 BACKGROUND AND RELATED WORK

In this section we discuss existing techniques and new ones that we implemented, inspired by testing approaches in other domains, for smart contract testing. We also discuss related work in measuring effectiveness of testing in terms of code coverage and fault finding. Finally, we briefly present related work in verification of smart contracts.

2.1 Existing Solidity Testing Approaches

We found four testing tools in the literature that use BF or AF techniques. We summarise the tools for these two techniques below.

Blackbox Fuzzing (BF) . is a simple technique that generates random test data according to a distribution for the different inputs [17]. This technique can generate input for all kinds of programs by generating a bit stream for representing the required data types. Three out of the four testing tools in the literature use BF. ContractFuzzer [25] takes ABI and bytecode files generated by the Solidity compiler as inputs and generates inputs using blackbox fuzzing. Echidna [13] is a Haskell library designed for property-based testing. It uses grammar-based fuzzing to generate test inputs based on user provided predicates or test functions. Writing the predicates and test functions requires significant expertise and is time consuming. SolAnalyser [2] generates random test inputs for detecting security vulnerabilities. For ease of representation and comparison, we pick a single BF implementation to use in our experiments – SolAnalyser [2] as it achieved the best average fault finding over the contracts in our dataset compared to other BF tools and it was easy to use with a well maintained implementation.

Adaptive Fuzzing (AF). refers to the technique used by sFuzz [34], that combines the fuzzing strategy in American Fuzzy Lop [47] (AFL) with a search-based technique for selecting seeds. The objective used in selecting the seeds is a quantitative measure (distance) on how far a seed is from covering any just-missed branch (one control flow edge away from covered node) . The technique evolves the initial random population by selecting test cases that cover a new branch and selecting one test case for each just-missed branch that is closest according to their distance metric. sFuzz keeps the number of test inputs generated small by only keeping the best seed for each just-missed branch. Overhead is reduced by only computing distance to just-missed branches, not all uncovered branches.

2.2 New Solidity Testing Approaches

Coverage-Guided Fuzzing with SMT solver (GF). uses a combination of blackbox and whitebox fuzzing. It first generates random inputs for a predetermined time and measures code coverage achieved. It then switches to whitebox fuzzing to gather path conditions for uncovered conditions. These constraints for uncovered regions are given to an SMT solver which returns inputs exercising feasible paths [10, 39].

The approach for GF is not new, and has been performed extensively in other domains [4, 24, 26, 39]. There is no existing tool for smart contracts that uses GF, and our implementation is the first in this domain.

Genetic Algorithm (GA) . finds the desired solution by using fitness functions and genetic operations: crossover and mutation [27, 45]. Fitness functions typically used in test generation are maximising coverage, minimising execution time (cost). The genetic operation, crossover, is a process of taking two existing test inputs (parents) and producing a new test input by combining them (offspring). The mutation operator modifies one or more test inputs in an individual according to a given probability. The goal of the mutation process is to increase the diversity of the population and to reduce similarity between individuals. Genetic algorithms have proven effective for testing Java programs [19], mobile [7, 31, 32], web [3, 14] and Internet of Things (IoT) applications [5] among others.

To our knowledge, there is no existing tool implementing GA for testing smart contracts. The implementation in this paper is a first for Solidity contracts.

Summary. Fuzzing techniques like BF rely almost entirely on random mutations to detect security vulnerabilities. Although proven to be powerful in revealing security bugs in real-world software [22, 30, 47], they are unlikely to reach certain parts of the code and have trouble reaching deeper branches. Approaches like GF, AF and genetic algorithm combine random mutations with guided input selection to avoid this problem. We aim to compare the effectiveness of the four techniques with respect to code coverage, fault detection and overhead.

2.3 Measuring Test Effectiveness

Effectiveness of testing has been traditionally measured in terms of (i) code coverage achieved, and (ii) fault finding [23, 38]. There is limited existing work in defining and measuring Solidity specific code coverage. We found 3 pieces of work – (1) Path coverage measurement proposed by Fu et al. [21] in 2019, (2) Statement and branch coverage measurement by Brownie [9], (3) Statement coverage implemented by solidity-coverage [41]. Fu et al. measure path coverage for symbolic execution by gathering sequences of opcode and branch conditions. Measuring path coverage incurs high overhead and their implementation is not easy to use with test inputs from other techniques. `Brownie` and `solidity-coverage` tools assess statement or opcode coverage (and branch coverage with Brownie) by instrumenting the abstract syntax tree followed by analysing the execution trace to detect the opcodes and branches executed. Our approach for coverage measurement is similar but we measure call coverage and event coverage in addition to opcode and branch coverage.

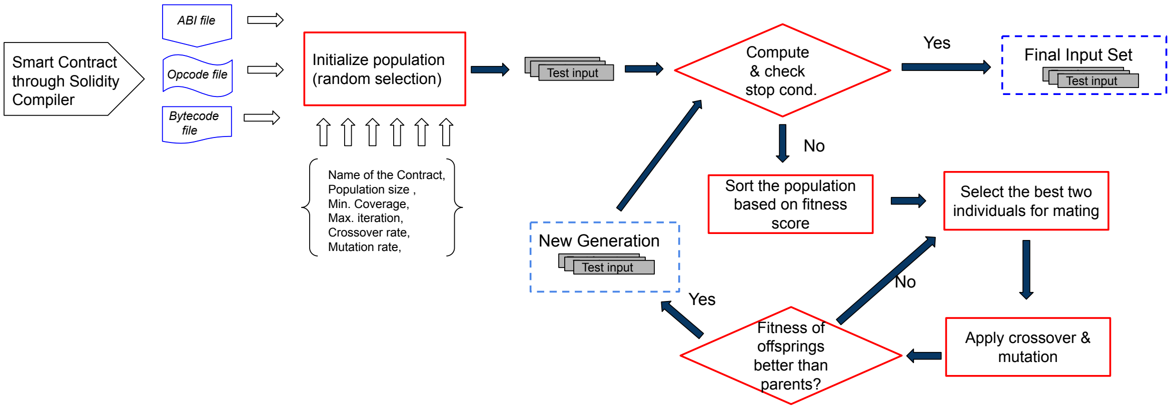

ABI file
Opcode file
Bytecode file
Smart Contract through Solidity Compiler
Initialize population (random selection)
Name of the Contract, Population size , Min. Coverage, Max. iteration, Crossover rate, Mutation rate,
Test input
Compute & check stop cond.
Yes
Final Input Set
Test input
No
Sort the population based on fitness score
Select the best two individuals for mating
New Generation
Test input
Yes
No
Fitness of offsprings better than parents?
Apply crossover & mutation

Figure 1: Genetic Algorithm working scheme

To measure fault finding, we use both artificially seeded (mutants) and real faults in our evaluation. Artificially seeding faults has been explored previously for Solidity contracts. Eth-mutants [8] is an open-source mutant generator for Solidity contracts. However, the mutation types supported are limited. It only performs replacement of <= to < and >= to >. MuSC [28] and MuContract [2] are mutation tools that support a wider range of mutation types, including ones specifically targeting Solidity syntax. Both mutation tools operate at the AST level. The primary difference between them is that MuSC makes every possible change in the AST for a given mutation type. For instance, for a relational operator mutation type, if there are 100 possible changes of this type in the original contract, then MuSC will generate 100 mutants covering all locations and options. MuContract, on the other hand, selects one change randomly among all possibilities of a relational operator change in the AST. It then generates one mutant for that mutation type. We use MuContract in our experiment to generate mutants for different vulnerability types to keep our dataset tractable. For fault finding with real faults, we use a dataset of 90 vulnerable contracts gathered by Durieux et al. [16]. Durieux et al. manually tagged the lines that contain the vulnerability for all 90 contracts facilitating the assessment of fault finding.

3 IMPLEMENTATION

In this Section, we discuss the following, (1) implementations of the four Solidity testing techniques, (2) definition and measurement of four Solidity code coverage metrics, and finally, a (3) fault seeding framework for different types of vulnerabilities. We support Solidity compiler versions - 0.4.25 and 0.5.8 to compile the contract code in our experiment for Ethereum Virtual Machine (EVM) version 3.0.0. Our implementations for testing techniques, code coverage and fault seeding are publicly available at https://github.com/doubleblind-conf/empirical_evaluation_for_sc.

3.1 Testing Techniques

3.1.1 BF. As mentioned in Section 2, we use an existing BF tool, SolAnalyser [2], in our experiments. BF implementation in SolAnalyser framework takes two arguments as inputs – the Application Binary Interface (ABI), and a bytecode file. These are generated from the Solidity contract with the Solidity compiler. ABI file holds information regarding functions in the contracts such as function name, function type, input types, and output types. Bytecode file holds the predefined bytecode of the smart contract. SolAnalyser implementation supports all Solidity types such as signed/unsigned integers types with widths ranging from 8 to 256 bits. The generated inputs call each of the functions in the smart contract and are written into a JSON file.

3.1.2 GA. Figure 1 shows the working scheme of the GA technique we implemented. We use Java for our implementation and take nine arguments as inputs. Four out of the nine input arguments are ABI, name of the contract, bytecode and opcode files. Opcode file holds the predefined opcode of the smart contract. We use these four arguments to generate the first random population. The remaining five arguments are directly related to GA operators – namely, size of the population, minimum desired coverage, number of iterations, mutation rate and crossover rate. Stopping condition for evolution is a pre-defined time of 15 seconds in our experiment. The reason to set 15 seconds for test generation time was inspired by time limit set within existing input generation tools [2, 25]. The papers report total experiment time and based on the number of contracts in their dataset, we calculate average test generation time to be around 25 to 30 seconds per contract. We fixed the limit at a slightly lower 15 seconds per contract to allow the experiments to be tractable. We use single-point crossover, along with user-defined crossover and mutation rate in our implementation. We ran our GA technique using different mutation and crossover rates ranging from 0.1 to 0.4 for each. We found the best mutation and crossover rate to be 0.2 based on opcode coverage achieved. The crossover operator creates two offspring test inputs, O1 and O2 from parent test inputs, P1 and P2. A value between 0 to 1 is chosen as the crossover rate, α. The first offspring contains $\alpha \times |P1|$ test inputs from P1 and (1 - α) $\times |P2|$ test inputs from P2. The second one contains $\alpha \times |P2|$ test inputs from P2 and (1 - α) $\times |P1|$ test inputs from P1. After the offsprings are generated, they are mutated by randomly updating some of the input values based on a user-defined mutation rate (between 0 and 1) that defines the extent to which the offspring is changed. Resulting offsprings are added to the new generation if they improve the fitness score achieved by their parents. We used opcode coverage as the fitness score. We measure opcode coverage of a candidate test input with our coverage measurement framework discussed in Section 3.2. We sort the population based on opcode coverage using fast-sorting dominance algorithm [12]

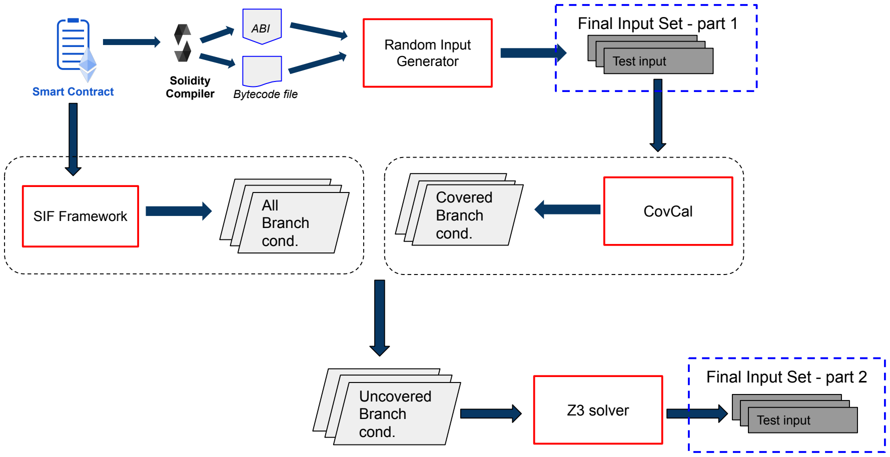


Figure 2: Coverage-Guided fuzzing with Z3 constraint solver

with time complexity $O(MN^2)$, where N is the population and M is the number of objectives. After sorting, we select the top two test inputs to be parents for mating. We then apply crossover and mutation to generate offsprings. If any of the offspring test inputs achieve better fitness score than their parent test inputs, we add them to the population. We maintain population size by removing an equal number of poorly performing test inputs. We repeat this process until the stopping condition is met.

3.1.3 GF with Z3 constraint solver. Figure 2 shows the workflow of our GF test generation technique. We first generate random inputs for a fixed time (half the input generation time in our experiments) and measure branch coverage achieved using the coverage measurement tool, presented in Section 3.2. For the remaining half of the time, we gather path constraints for uncovered branch conditions and feed them to the Z3 SMT solver. The solver returns inputs for uncovered branches that can be triggered. A Python script is then used to analyse uncovered branches and synthesise a Z3 program containing constraints to cover those branches. Finally, the Z3 program produces inputs for covering feasible conditions. The initial random inputs along with inputs from Z3 form the final input set.

Consider the following example contract named *Magi*.

Listing 1: setValue function from the Magi smart contract

```
function setValue(uint idx, uint newValue){
    if (idx == 0) {
        do something...
    } else if (idx == 1) {
        do something...
    } else if (idx == 2) {
        do something...
    } else if (idx == 3) {
        do something...
    } else {
        revert();
    }
}
```

With BF, we sample inputs from the range of a 256 bit unsigned integer. Majority of the random inputs execute the last `else` branch, since the likelihood of choosing values 0, 1, 2 and 3 in this range is low. GF technique, on the other hand, generates constraints to be solved for each condition checking *idx* variable values. The example below shows the contraint gathered by the instrumentation when visiting the function AST for the first if condition checking *idx* == 0.

Listing 2: Constraints generated

```
#Initiate the Z3 solver
S = solver()
#Declare idx variable in Z3
#The UInt type is a customised Z3 type
idx = UInt('idx')
#Add the contraint to the solver
s.add(idx == 0)
#Solve the constraint
s.check()
```

For all the conditions checking `idx` values, a unique Python script, resembling the above listing, is generated with different constraints on the values for `idx` corresponding to respective conditions. These constraints are then fed to the Z3 solver which in turn generates inputs for feasible conditions.

3.1.4 AF. We used the sFuzz tool implemented by Nguyen et al [34] as AF in our experiments. The source code of the tool is publicly available at https://github.com/duytai/sFuzz. We set the time for input generation in the sFuzz implementation to 15 seconds, matching the time set in the other tools.

3.2 Solidity Code Coverage

In this Section, we discuss the coverage measurement framework, `CovCal`, and the code coverage metrics we use in our measurement.

3.2.1 CovCal implementation. Executing the Solidity contract with an input in the EVM produces an execution trace. We implemented `CovCal` in nodejs v8.11.3 as an extension to EVM to analyse the execution trace produced by EVM. We capture both the execution result and progression steps returned by EVM. The execution result shows the return value of the function, validation code - 0 or 1, gas usage, gas refund, among other information. The progression step shows the execution trace in the form of runtime opcodes, similar to assembly code. The recorded execution result is used to determine gas usage. The recorded progression steps are used for calculating coverage metrics defined below,

3.2.2 Opcode Coverage. Defined as fraction of opcodes executed by test inputs to total number of opcodes in the contract. To measure this, we record the opcodes executed at runtime. We removed duplicate opcode sequences that may occur within loops or repeated function calls. Total number of opcodes in the contract is inferred using the Solidity compiler. We calculate the opcode coverage achieved by a test input or input set using the following formula,

$$Opcode\ Coverage\ = \frac{\#Opcodes\ executed}{Total\ \#opcodes\ in\ contract} \tag{1}$$

`Brownie` [9] and `solidity-coverage` [41] use the same definition for Opcode coverage.

3.2.3 Branch Coverage. Measured as number of branches covered by the input set to total number of branches in the contract. We first compile a list of all the branches in the contract. Every branch executed by a test input in the input set is marked as covered. We then calculate branch coverage achieved by an input set using the following formula,

$$Branch\ Coverage\ = \frac{\#Covered\ branches}{Total\ \#branches\ in\ contract} \tag{2}$$

Branch coverage measured by `Brownie` [9] matches this definition.

3.2.4 Event Coverage. Measured as number of executed events to total number of events in the contract. For event coverage calculation, we first record opcodes executed by all the test inputs in the input set. From these recorded opcodes, we retrieve 'LOG' opcodes, since EVM will return this opcode whenever an event is triggered [46]. We calculate event coverage achieved by an input set using the formula below.

$$Event\ Coverage\ = \frac{\#Executed\ events}{Total\ \#events\ in\ contract} \tag{3}$$

3.2.5 Call Coverage. Defined as number of calls executed by test inputs in the input set to the total number of calls present in the contract. We record runtime opcodes executed and gather the ones that correspond to 'CALL'. Etherum yellow paper [46] confirms EVM will return this opcode whenever a call is triggered. We calculate call coverage using the formula below.

$$Call\ Coverage\ = \frac{\#Executed\ calls}{Total\ \#calls\ in\ contract} \tag{4}$$

We are not aware of any other existing tool supporting event or call coverage for Solidity code.

3.3 Fault Seeding

Fault seeding is also referred to as mutant generation where a mutant is a faulty contract with a single seeded fault. We use two existing tools for seeding faults in Solidity contracts - `MuContract` [2] and `Mutec` [36]. `MuContract` exists as part of the SolAnalyser framework [2] to seed Solidity-specific vulnerabilities. `Mutec` is a tool for seeding operator faults in C/C++ programs [36]. We extend `Mutec` to support operator mutations in Solidity contracts as `MuContract` does not support this mutation type.

We generate ten mutations of different types for each Solidity contract. Seven out of the ten mutation types are specific to Solidity syntax. We use the `MuContract` tool to generate these mutation types, namely – integer overflow, integer underflow, division by zero, timestamp usage, tx.origin usage, unchecked send, repetitive function call [2]. The remaining three mutation types were traditional operator-related mutations for relational, logical and arithmetic operators. For these, we modified the `Mutec` tool to support Solidity syntax. `Mutec` is based on the Clang LibTooling framework that recursively visits the AST of C and C++ programs to seed faults of a given type. To support Solidity, we combine SIF [37], a Solidity instrumentation framework, with `Mutec`. `Mutec` marks locations of relevant operators for arithmetic, logical and relational mutations while SIF traverses the AST. Once SIF finishes visiting the entire AST, `Mutec` generates mutations for relevant operators in Solidity contracts.

The ten mutation types used in our fault seeder are inspired from known attacks on Ethereum smart contracts listed in [33]. We also support reentrancy mutations in our fault seeder implementation. However, our execution environment does not provide multiple threads to help simulate this attack. We, therefore, do not report on reentrancy mutations in our experiment.

For a given contract, fault finding ability of an input set is measured as

$$Fault\ Finding\ Score\ = \frac{\#Vulnerabilities\ Killed}{Total\ \#Vulnerabilities} \tag{5}$$

Execution environment. helps determine the fault finding score for mutations and real vulnerabilities. Our execution environment is implemented as an extension to EVM to analyze the execution traces and report triggered vulnerabilities. We used external `node.js` modules to combine EVM and execution trace analysis. Execution traces from EVM are in the form of opcode sequences. To determine if a mutation is killed, we first run the original and mutated contract in the execution environment. We then compared their execution results, which comprise function return values, validation code, gas usage, and refund. Comparing execution results will suffice for certain mutation types like integer overflow, integer underflow, division by zero, arithmetic, logical, relational. For vulnerable contracts where the original correct contract is unavailable, we monitor the execution and compute expected results using an external `big-integer` module in our execution environment that provides a larger value range to help avoid overflow or underflow in arithmetic operations.

For other mutation types, such as timestamp-usage, tx. origin, unchecked send and repetitive call, execution results between original and mutated contracts remain unchanged. For these types, we analyze the execution traces (opcode sequences) to detect mutations or vulnerabilities. For example, If we find a `Call` operation in an execution trace with no matching `Revert`, we signal the presence of an unchecked send vulnerability. If the vulnerability detected matches the description of a seeded fault or known vulnerability, we mark it as killed/detected.

4 EXPERIMENT

We evaluate effectiveness of the four testing techniques using two data sets - (1) *Random-C* - 1665 randomly sampled contracts from Etherscan that are assumed correct with no known vulnerabilities, and (2) *Vulnerable-C* - 90 contracts with known vulnerabilities from the SB curated dataset [43]. We fix input generation time at 15 seconds per contract for each of the four testing techniques. The time we used is comparable to input generation times in other contract testing papers [2, 25]. We investigate the following research questions:

Q1. Code Coverage comparison: *Which input generation technique achieves best code coverage on average?*

To answer this question, we generated input sets using BF, GF, AF and GA for the contracts in the Random-C and Vulnerable-C datasets. We then ran the generated input sets and measured branch coverage, opcode coverage, event coverage and call coverage using the `CovCal` framework.

Q2. Fault finding comparison: *Which test generation technique is most effective at fault finding on average?*

To answer this question, we assess fault finding using Random-C and Vulnerable-C datasets. Since contracts in Random-C have no known vulnerabilities, we seed 10 artificial faults (or mutants) in each of the 1665 contracts by analysing the source code and applying mutation operators to eligible locations using MuContract and Mutec. Fault finding for a input set is measured as fraction of mutants killed (for seeded faults) or vulnerabilties detected (for real faults). We compare average mutation score achieved by the techniques across all 1665 contracts in the Random-C dataset and average fault finding across all 90 contracts in the Vulnerable-C dataset. Contracts in the Vulnerable-C dataset have been manually annotated (by the dataset creators) with the type of vulnerability and its location in the contract. When a test execution in our environment encounters a vulnerability, information on its location and type appears in the error message. We match this to the annotation in the dataset to mark the vulnerability as detected.

Q3. Input set size and execution time: *What is the size of the input set generated by each technique and how long does it take to run?*

We measured size of the input sets generated by BF, GF, AF and GA for each Solidity contract in the Random-C and Vulnerable-C datasets. We ran the test inputs on EVM to measure average execution time over 10 repeated runs of each input set. We used the same machine (Intel Quad Core i5-5200 CPU 2.20GHz, 8GB DDR3, 64-bits) for all runs. We report average input set sizes and average input set execution times across all contracts in the Random-C and Vulnerable-C datasets.

4.1 Data Set

Random-C . We collected 1665 verified[3] smart contracts of different sizes from Etherscan [18], ranging from 49 to 2856 LOC. We implemented a web-crawler to collect smart contracts whose source codes are publicly available on Etherscan web-site. We restricted the Random-C dataset to 1665 contracts in the interest of experiment resources and time as each of these contracts had a further 10 mutated contracts for the fault finding experiment. The contracts in our dataset covered a wide range of Solidity language features - 77 % of the contracts contain branch statements, 91% contain event usages, 57% comprise external calls, and finally, 27% contain loops.

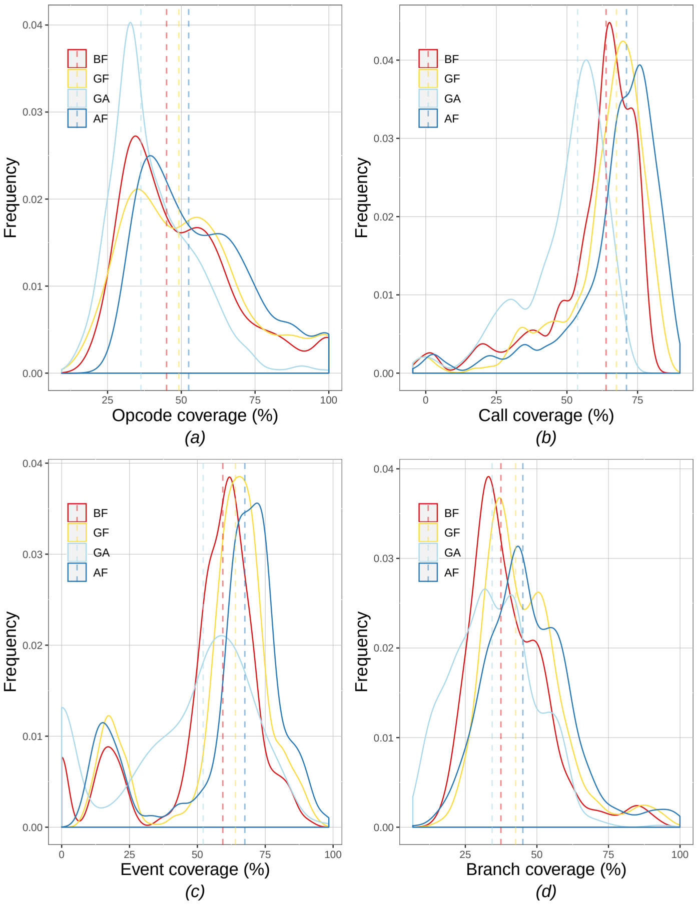


Figure 3: Histogram frequencies of coverage achieved by BF, GF, AF and GA input sets for 1665 Solidity contracts in the Random-C dataset

Vulnerable-C. Durieux et al. [16] created the SB Curated dataset that contains Solidity contracts deployed in the Ethereum network with known vulnerabilities. Contracts in the dataset have been manually tagged with the location and type of its vulnerability. We used 90 vulnerable contracts with 5 different vulnerability types, namely- tx.origin usage (18) , integer overflow (7), integer underflow (8), timestamp usage (5), unchecked send (52).

Dataset Availability. All the contracts in our datasets, implementation of testing techniques, coverage measurement and fault seeding, along with scripts to run the experiments can be found at https://github.com/doubleblind-conf/empirical_evaluation_for_sc.

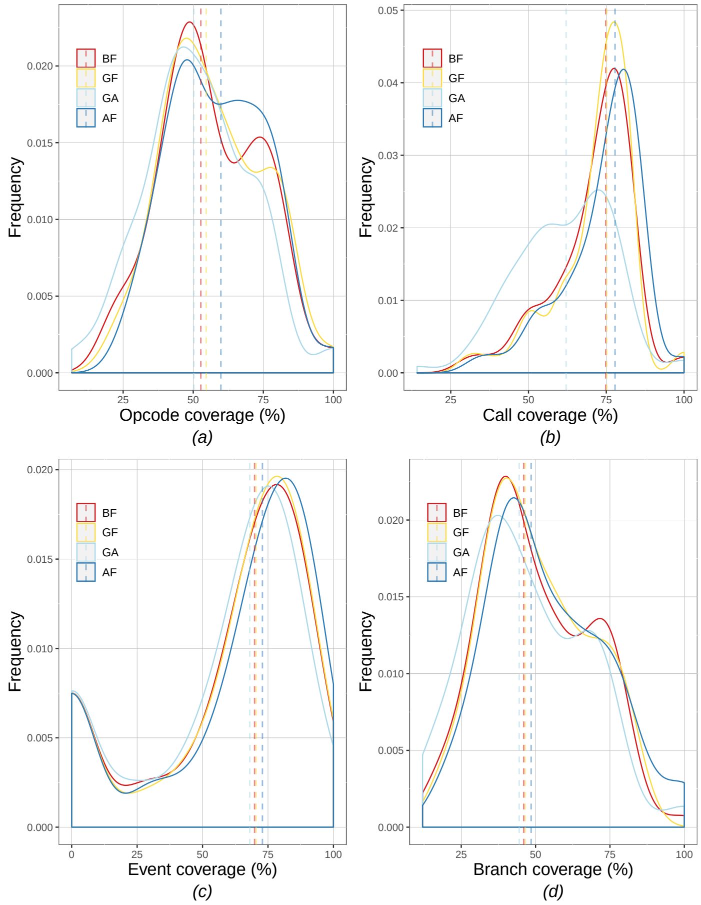
5 RESULT

In this Section, we report and discuss results in the context of the research questions presented in Section 4.

5.1 Q1. Code Coverage Comparison

We present opcode, call, event and branch coverage achieved by BF, GF, AF and GA input sets over the 1665 Solidity contracts in the

[3] Source code and byte code of the contract conform with each other.



(a)

(b)

(c)

(d)

Figure 4: Histogram frequencies of coverage achieved by BF, GF, AF and GA input sets for 90 Solidity contracts in the Vulnerable-C dataset

Random-C and 90 Solidity contracts in the Vulnerable-C datasets in Figures 3 and 4, respectively. We discuss performance of the techniques with respect to each of the coverage metrics in the Sections below. We check if differences between the techniques is significant using one-way Anova and Tukey’s honest significant difference (HSD) test. P-values for pairwise comparison of techniques using this test is shown in Tables 2 and 3 for the Random-C and Vulnerable-C datasets, respectively, with emphasis on values that show significant difference (at 5% significance level).

Uncovered Code. Median coverage achieved by all four input generation techniques is not that high (ranging from 31% to 78%) for contracts in both datasets in Figures 3 and 4. For instance, opcode coverage in Figures 3(a) and 4(a) show that more than 50% of the opcodes on average remain uncovered by all techniques. The main reasons for the large proportion of uncovered regions observed with all techniques are,

1. Restricted input generation time - In our experiment, we restrict input generation time with each technique to 15 seconds. Increasing the time for input generation will result in better coverage as seen in Table 1 which shows the average opcode coverage (over a smaller sample of 150 contracts) achieved by each technique with different generation times.

Generation Time	BF	GF	AF	GA
15 secs	45.1%	47.5%	52.4%	35.5%
30 secs	52.3%	55.8%	58.8%	47.9%
45 secs	57.7%	59.4%	65.1%	56.5%

Table 1: Average opcode coverage achieved with increasing test generation time on 150 contracts

Comparison	Opcode	Call	Event	Branch
GA vs BF	**0.041**	**0.028**	**0.021**	**0.048**
GA vs GF	**0.023**	**0.011**	**0.008**	**0.031**
GA vs AF	**0.016**	**0.001**	**0.004**	**0.007**
GF vs BF	0.554	0.511	0.178	0.478
GF vs AF	0.057	0.079	0.417	0.067
BF vs AF	0.050	0.050	0.051	**0.044**

Table 2: P-values using One way Anova Tukey’s HSD for pairwise comparison of coverage achieved for Random-C dataset

2. Dependency between functions not considered - All the techniques generate inputs to execute functions within a contract without considering dependency on other functions. As a result, conditions dependent on values from other functions may not be be satisfied. Statements depending on these conditions remain uncovered. An example is a smart contract with three dependent functions – `assign`, `approve`, and `send`. The `send` function sends money to the user address that is approved by the `approve` function which in turn checks if the address was allocated by the `assign` function. Thus for a test input to cover the send money transaction, the input will also need to have passed the condition checks in the `assign` and `approve` functions. Such a constraint dependent on other functions is not considered by any of the testing techniques.

3. Unused Functions in Inherited Contracts - Solidity language supports inheritance using the keyword `is`. Inherited contracts are found in both Random-C and Vulnerable-C datasets. We find not all inherited functions get called in the child contract. These uncalled functions contribute to the reduced coverage observed.

4. Modifier usage in the function - According to Solidity 0.6.2 documentation [15], "Modifiers can be used to change the behaviour of functions in a declarative way." Listing 3 shows a code snippet for a modifier function from the Bitconnect contract in the Random-C dataset that restricts calling of this function to the creator. None of the input generation techniques consider conditions imposed by function modifiers which in turn affects code coverage.

Listing 3: onlyCreator modifier from the Bitconnect smart contract in Random-C dataset

```
modifier onlyCreator() {
    require(msg.sender == creator);
    _;
}
```

5.1.1 Branch Coverage. Figures 3(d) and 4(d) show the histogram frequencies of branch coverage achieved by all four techniques over the Random-C and Vulnerable-C datasets, respectively. Median coverage is shown as a vertical dashed line. Median branch coverage for contracts in the Random-C dataset are 45% for AF,

Comparison	Opcode	Call	Event	Branch
GA vs BF	**0.049**	**0.001**	**0.065**	**0.049**
GA vs GF	**0.035**	**0.001**	0.073	**0.049**
GA vs AF	**0.008**	**0.001**	**0.014**	**0.011**
GF vs BF	0.719	0.758	0.571	0.763
GF vs AF	0.075	0.121	0.056	0.091
BF vs AF	0.050	0.205	0.061	0.099

Table 3: P-values using One way Anova Tukey's HSD for pairwise comparison of coverage achieved for Vulnerable-C dataset.

43% for GF, 37% for BF and 35% for GA. Median values over the Vulnerable-C dataset are 49% for AF, 47% for GF, 46% for BF and 44% for GA. Reasons for uncovered branches observed with all the technqiues was discussed earlier in the context of uncovered code, namely, restricted input generation time, function dependencies and modifier usage. One-way Anova and Tukey's HSD test revealed significant difference between branch coverage achieved by fuzzing techniques (BF, GF and AF) versus the GA technique. A significant difference was also observed for BF versus AF over the contracts in the Random-C dataset, not the Vulnerable-C dataset. This is because nested conditions that BF struggles to cover occurs more frequently in the Random-C (29%) dataset than in Vulnerable-C (13%).

5.1.2 Opcode Coverage. Figure 3(a) and Figure 4(a) shows histogram frequencies of opcode coverage achieved by the four techniques over the Random-C and Vulnerable-C datasets, respectively. For the Random-C dataset, medians observed were 52% for AF, 49% for GF, 45% for BF and 36% for GA. Median coverage values over the Vulnerable-C dataset were – 60% for AF, 55% for GF, 53% for BF and 50% for GA.

Tables 2 and 3 show there is a significant difference between the fuzzing techniques versus the GA technique in opcode coverage achieved. On the other hand, pairwise comparison of AF, GF, BF revealed no significant difference.

It is worth noting for GF that several constraints provided to the Z3 solver for uncovered branches provide no improvement in opcode coverage or remain unsolved. The first scenario of no improvement occurred with constraints for uncovered false branches of conditions with no code within, as seen in the following example.

Listing 4: btcToTokens function from the Bitconnect smart contract

```
function c(uint _0xbtcAmount) {
    some preparation...
    if (_0xbtcAmount != 0) {
        do something...
    }
}
```

Else statements are often omitted to limit gas usage in contracts. Many contracts with if-conditions resemble the example in the above listing, with no error-handling or code in the false branch. As a result, covering these additional branches do not result in additional opcodes being covered. The second scenario of constraints remaining unsolved occurred when conditions included Solidity specific features like modifiers, block number, blockhash, address data type and its member functions `send` and `transfer`, etc. Listing 5 shows an example function with an if condition that uses block related dependencies and address data type that Z3 fails to handle.

Listing 5: clearApproval function from a smart contract in Random-C dataset

```
function clearApproval(address _owner, uint256 _tokenId) onlyOwner {
  if (ownerOf(_tokenId) == _owner && tokenApprovals[_tokenId] != address(0) && tokenApprovals[_tokenId].fromBlock > _blockNumber ) {
    tokenApprovals[_tokenId] = address(0);
  }
}
```

Z3 only solves 185 constraints from 54 smart contracts in our dataset. For these 54 smart contracts, opcode coverage increased from an average of 38.1% without using the solver to 54.3% with the solver.

5.1.3 Call Coverage. Figures 3(b) and 4(b) shows the call coverage frequency achieved by the four input generation techniques over the Random-C and Vulnerable-C datasets, respectively. Median call coverage values over the Random-C dataset are – 71% for AF, 68% for GF, 64% for BF and 54% for GA. Median call coverage values over contracts in the Vulnerable-C dataset are 78% for AF, 75% for GF, 75% for BF and 62% for GA. We find fuzzing techniques achieve significantly better call coverage over contracts in both datasets than the GA technique. No significant difference was noted between pairs of fuzzing techniques.

5.1.4 Event Coverage. Figures 3(c) and 4(c) shows event coverage achieved by the testing techniques over the Random-C and Vulnerable-C datasets. Random-C median values were 67% for AF, 64% for GF, 59% for BF and 52% for GA. Vulnerable-C median event coverage was 73% for AF, 70% for GF, 70% for BF and 68% for GA.

As with other coverage criteria, difference between event coverage achieved by fuzzing techniques and GA was statistically significant. As seen in Figures 3 and 4, median values for event and call coverage are higher than those observed for opcode and branch coverage with all four techniques. This is because many of the event and call operations in a contract are not embedded within a conditional statement (around 55% for Random-C contracts) allowing them to be easily reached.

Summary. Over both datasets and for all four coverage metrics, AF achieves the best median performance among the four input generation techniques. Coverage achieved by all four techniques over contracts in both datasets is not that high, with medians ranging from 31% to 78% owing to small input generation time of 15 seconds per contract and not considering function dependencies and modifiers. Performance of GF could be improved by using an SMT solver that handles Solidity specific constructs.

Median values for coverage achieved over the Vulnerable-C dataset is slightly better than the Random-C dataset since the contracts are smaller on average (101 LOC in Vulnerable-C versus 389 LOC for Random-C) and contain fewer branches on average per

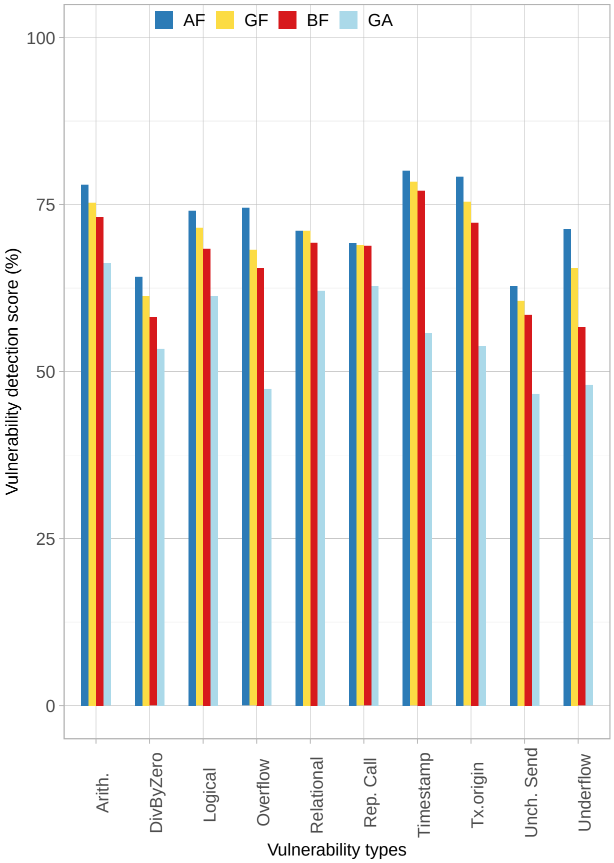


Figure 5: Fault finding score using 16650 artifical faults seeded in the Random-C dataset, grouped by mutation type

contract (10 branches per contract for Vulnerable-C versus 22 for Random-C).

5.2 Q2. Fault Finding Comparison

We assess fault finding achieved by the four testing techniques using (1) Artificial seeded vulnerabilties in the Random-C dataset, and (2) Known vulnerabilities in the Vulnerable-C dataset.

Artificial Faults. Figure 5 shows average mutation score for each technique grouped by mutation (or vulnerability) type. We seeded a single fault for each of the 10 different mutation types for each contract resulting in 16650 mutated contracts in our dataset. Across all mutated contracts, we find AF performs best at fault finding with an average mutation score of 72% versus 70% for GF (p-value = 0.39) and 67% for BF (p-value = 0.28) with no significant difference between them confirmed with one-way Anova followed by Tukey HSD test. However, AF was significantly better than GA with 56% average mutation score (p-value = 0.03). We also compared the average mutation score achieved by the four techniques for each of the different mutation types. P-values with one way Anova and Tukey HSD test is provided in Table 4 with emphasis on values showing significant difference (at 5% significant difference).

Overall, we find fuzzing techniques outperform the GA technique across all mutation types. As seen in Section 5.1, fuzzing techniques (AF, GF, BF) achieve better coverage and therefore execute more statements and operations than the GA technique. This in turn helps fuzzing techniques exercise and reveal more mutations than GA. For relational and repetitive function call mutation types, AF, BF, GF and GA have comparable perfomance with no significant difference, as none of them specifically target these constructs in the contract.

Finally, we observe 20–38% of the mutants remain alive even with the most effective testing technique, AF, in our experiment. This is because the generated inputs do not cover approximately 40% of the code and, as a result, fail to reveal seeded faults in these uncovered code regions. Reasons for regions of code being uncovered with the testing techniques was discussed in Section 5.1.

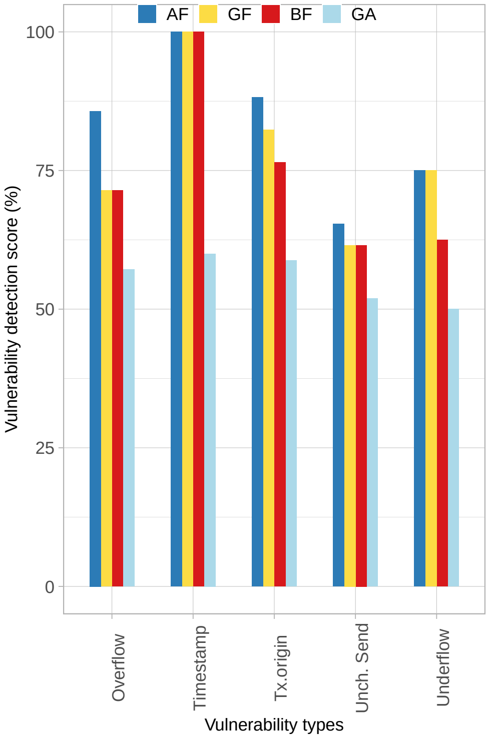


Figure 6: Fault finding score with 90 known vulnerabilities in the Vulnerable-C dataset, grouped by vulnerability type

Real Faults. Across all 90 vulnerable contracts in the Vulnerable-C dataset, we find AF performs best at vulnerability detection with an average score of 83% versus 78% for GF, 74% for BF and 56% for GA. One-way Anova test shows a significant difference between performance of AF and GA. Figure 6 shows average vulnerability detection score for each technique grouped by vulnerability type. AF has highest fault finding score for all vulnerability types, similar to our observation over artificial faults. All the fuzzing techniques achieve 100% detection score for the `timestamp` vulnerability. Presence of this vulnerability in mostly small contracts (37 LOC on average) and lack of conditional statements around many of them made it easy to detect this vulnerability type.

We find all four techniques were unable to detect vulnerabilities in 17 out of the 90 contracts. This may be attributed to the following reasons, (1) Uncovered code - vulnerabilities in uncovered code remain undetected. As seen earlier, uncovered code is a pervasive issue across our contracts. (2) Branch conditions in contracts that

Pairwise Comp.	Arithmetic	Div by Zero	Logical	Overflow	Relational	Rep. call	Timestamp	Tx. origin	Unchecked send	Underflow
GA vs BF	0.051	0.057	0.173	**0.005**	0.097	0.213	**0.004**	**0.017**	**0.044**	**0.039**
GA vs GF	**0.041**	**0.044**	**0.049**	**0.001**	0.074	0.179	**0.004**	**0.009**	**0.037**	**0.011**
GA vs AF	**0.037**	**0.041**	**0.036**	**0.001**	0.071	0.103	**0.001**	**0.001**	**0.033**	**0.002**
GF vs BF	0.144	0.471	0.365	0.515	0.736	0.716	0.469	0.148	0.214	**0.046**
GF vs AF	0.333	0.098	0.111	0.634	0.971	0.918	0.338	0.113	0.377	0.051
BF vs AF	0.071	0.115	0.332	0.071	0.741	0.883	0.377	0.092	0.125	**0.034**

Table 4: P-values using One way Anova Tukey's HSD for pairwise comparison of mutation score.

involve Solidity specific variable types such as deployed address, balance, remain uncovered by test inputs from all four techniques. Listing 6 shows an example of an unchecked send vulnerability in the uncovered true branch of the if statement with Solidity specific features – block number and contract balance.

Listing 6: finalize function from the MigrationAgent smart contract

```
function finalize(uint _value) {
...
if(this.balance> _value && allowed[from
     ][msg.sender]>=_value && _value > 0
     && Approval[from]._fromBlock >
     _blockNumber)
     owner.send(_value);
     // Unchecked send vulnerability
...
}
```

Summary. We find fault finding scores for all four techniques is higher for real over artificial vulnerabilties. This is because all four testing techniques achieve better code coverage over contracts in the Vulnerable-C dataset compared to Random-C dataset.

Tech.	Average input set size		Average Execution Time (secs)	
	Random-C	Vulnerable-C	Random-C	Vulnerable-C
BF	973	1321	7.93 secs	9.02 secs
GF	436	587	4.31 secs	4.78 secs
AF	47	24	0.8 secs	0.31 secs
GA	100	100	1.78 secs	1.65 secs

Table 5: Average input set size and execution time per contract

5.3 Q3. Input set size and execution time of techniques.

For the Random-C and Vulnerable-C datasets, we present average size of the input sets generated by the four testing techniques and their execution times in Table 5. BF, as one might expect, generates the largest input set in the given time of 15 seconds - an average of 973 test inputs for Random-C contracts and 1321 test inputs for Vulnerable-C contracts. GF generates half as many test inputs as BF. This is because GF runs random input generation for half the generation time and the remaining time is used by the Z3 solver to generate inputs for uncovered branch conditions. The Z3 solver does not generate many test inputs as it was only able to solve constraints for 54 contracts. AF generates a much smaller input set, 47 test inputs and 27 test inputs on average, for contracts in the Random-C and Vulnerable-C datasets, respectively. This is because the AF algorithm reduces the input set size with respect to branch coverage after every evolution step. Input set size for GA remains constant at 100 as that is the population size provided. Trends in execution time follow input set size with BF taking up most time and AF with least time. Overall AF generates the smallest input set on average with least overhead in execution.

5.4 Threats to Validity

A potential threat to the internal validity is bugs in implementation of the testing techniques or coverage measurement. We extensively tested our implementatons and manually inspected them to mitigate this risk. Furthermore, the implementations for the testing techniques and coverage measurement along with the raw data are publicly available for other researchers and potential users to check the validity of our results.

A potential threat to the external validity is related to the fact that the set of smart contracts we have considered in this study may not be an accurate representation of a contract under test. We attempt to reduce the selection bias by leveraging a large collection of 1665 real, reproducible smart contracts. Additional threat to validity is caused by using artificially seeded faults to assess fault finding. To mitigate this threat, we also used a dataset of 90 vulnerable contracts, collected by Durieux et al. [16], deployed in the Ethereum network to assess fault finding. We also aim to reduce threats to external validity and ensure the reproducibility of our evaluation by providing the scripts used to run the evaluation, and all data gathered.

6 CONCLUSION

In this paper, we evaluated four smart contract input generation techniques - BF, GF, AF and GA by measuring, (1) Different types of code coverage - branch, event, call and opcode coverage, and (2) Fault finding - using real and artificially seeded vulnerabilities.

We used 2 datasets – Random-C, with 1665 contracts from Etherscan, and Vulnerable-C comprising 90 contracts with known vulnerabilities. The fuzzing techniques did significantly better than GA at achieving code coverage and fault finding for a fixed input generation time of 15 seconds per contract. AF did slightly better than BF and GF in terms of median coverage and fault finding values but the difference was not statistically significant. AF had the least overhead among the four techniques, generating the smallest input sets while being effective at covering the code and detecting vulnerabilities.

There is room for improvement for all four techniques in terms of code coverage and fault finding. We believe considering dependencies between functions and handling Solidity specific features will help improve the performance of these techniques considerably.

REFERENCES

[1] Moataz A Ahmed and Irman Hermadi. 2008. GA-based multiple paths test data generator. *Computers & Operations Research* 35, 10 (2008), 3107–3124.

[2] Sefa Akca, Ajitha Rajan, and Chao Peng. 2019. SolAnalyser: A Framework for Analysing and Testing Smart Contracts. In *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 482–489.

[3] Nadia Alshahwan, Xinbo Gao, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, Taijin Tei, and Ilya Zorin. 2018. Deploying search based software engineering with Sapienz at Facebook. In *International Symposium on Search Based Software Engineering*. Springer, 3–45.

[4] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. 2012. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. 1–11.

[5] Parvaneh Asghari, Amir Masoud Rahmani, and Hamid Haj Seyyed Javadi. 2019. Internet of Things applications: A systematic review. *Computer Networks* 148 (2019), 241–261.

[6] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A survey of attacks on ethereum smart contracts (sok). In *Principles of Security and Trust*. Springer, 164–186.

[7] Catherine A Bliss, Morgan R Frank, Christopher M Danforth, and Peter Sheridan Dodds. 2014. An evolutionary algorithm approach to link prediction in dynamic social networks. *Journal of Computational Science* 5, 5 (2014), 750–764.

[8] Federico Bond. [n.d.]. A mutation testing tool for Solidity contracts . https://github.com/federicobond/eth-mutants,Accessed on: April 19, 2020.

[9] Brownie. [n.d.]. Brownie. https://github.com/iamdefinitelyahuman/brownie-v2, Accessed on: August 24, 2020.

[10] Chen Chen, Baojiang Cui, Jinxin Ma, Runpu Wu, Jianchao Guo, and Wenqian Liu. 2018. A systematic review of fuzzing techniques. *Computers & Security* 75 (2018), 118–137.

[11] Huashan Chen, Marcus Pendleton, Laurent Njilla, and Shouhuai Xu. 2019. A Survey on Ethereum Systems Security: Vulnerabilities, Attacks and Defenses. *arXiv preprint arXiv:1908.04507* (2019).

[12] Keith D Cooper, Timothy J Harvey, and Ken Kennedy. 2001. A simple, fast dominance algorithm. *Software Practice & Experience* 4, 1-10 (2001), 1–8.

[13] crytic. [n.d.]. Ethereum fuzz testing framework. https://github.com/crytic/echidna,Accessed on: April 19, 2020.

[14] Giuseppe A Di Lucca and Anna Rita Fasolino. 2006. Testing Web-based applications: The state of the art and future trends. *Information and Software Technology* 48, 12 (2006), 1172–1186.

[15] Solidity Documentation. [n.d.]. Solidity in Depth âĂŤ Solidity 0.6.2 documentation. https://docs.soliditylang.org/en/v0.6.2/solidity-in-depth.html,Accessed on: February 25, 2021.

[16] Thomas Durieux, João F Ferreira, Rui Abreu, and Pedro Cruz. 2020. Empirical review of automated analysis tools on 47,587 Ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 530–541.

[17] Jon Edvardsson. 1999. A survey on automatic test data generation. In *Proceedings of the 2nd Conference on Computer Science and Engineering*. 21–28.

[18] Etherscan. [n.d.]. Etherscan - The Ethereum Block Explorer. https://etherscan.io/txs,Accessed on: April 19, 2020.

[19] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 416–419.

[20] Gordon Fraser and Andrea Arcuri. 2016. Evosuite at the sbst 2016 tool competition. In *Proceedings of the 9th International Workshop on Search-Based Software Testing*. 33–36.

[21] Menglin Fu, Lifa Wu, Zheng Hong, Feng Zhu, He Sun, and Wenbo Feng. 2019. A critical-path-coverage-based vulnerability detection method for smart contracts. *IEEE Access* 7 (2019), 147327–147344.

[22] The Peach Fuzzer Platform. [n.d.]. Integrating security into your DevOps Lifecycle GitLab. https://about.gitlab.com/solutions/dev-sec-ops/,Accessed on: February 24, 2021.

[23] Gregory Gay, Ajitha Rajan, Matt Staats, Michael Whalen, and Mats PE Heimdahl. 2016. The effect of program and model structure on the effectiveness of mc/dc test adequacy coverage. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 25, 3 (2016), 1–34.

[24] Karthick Jayaraman, David Harvison, Vijay Ganesh, and Adam Kiezun. 2009. jFuzz: A concolic whitebox fuzzer for Java. (2009).

[25] Bo Jiang, Ye Liu, and WK Chan. 2018. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 259–269.

[26] Su Yong Kim, Sangho Lee, Insu Yun, Wen Xu, Byoungyoung Lee, Youngtae Yun, and Taesoo Kim. 2017. Cab-fuzz: Practical concolic testing techniques for {COTS} operating systems. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*. 689–701.

[27] Manoj Kumar, Mohamed Husain, Naveen Upreti, and Deepti Gupta. 2010. Genetic algorithm: Review and application. *Available at SSRN 3529843* (2010).

[28] Zixin Li, Haoran Wu, Jiehui Xu, Xingya Wang, Lingming Zhang, and Zhenyu Chen. 2019. MuSC: A Tool for mutation testing of Ethereum smart contract. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1198–1201.

[29] Jie Liang, Yu Jiang, Mingzhe Wang, Xun Jiao, Yuanliang Chen, Houbing Song, and Kim-Kwang Raymond Choo. 2019. Deepfuzzer: Accelerated deep greybox fuzzing. *IEEE Transactions on Dependable and Secure Computing* (2019).

[30] Xinyao Liu, Baojiang Cui, Junsong Fu, and Jinxin Ma. 2020. HFuzz: Towards automatic fuzzing testing of NB-IoT core network protocols implementations. *Future Generation Computer Systems* 108 (2020), 390–400.

[31] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. Evodroid: Segmented evolutionary testing of android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 599–609.

[32] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 94–105.

[33] Dalal Nemin, Mueller Bernhard, and Lim Wanseob. [n.d.]. Ethereum Smart Contract Best Practices . https://consensys.github.io/smart-contract-best-practices/known_attacks/,Accessed on: January 30, 2021.

[34] Tai D Nguyen, Long H Pham, Jun Sun, Yun Lin, and Quang Tran Minh. 2020. sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 778–788.

[35] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2020. Parmesan: Sanitizer-guided greybox fuzzing. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 2289–2306.

[36] Chao Peng. [n.d.]. An automated mutation testing tool for C . https://github.com/chao-peng/mutec,Accessed on: May 10, 2020.

[37] Chao Peng, Sefa Akca, and Ajitha Rajan. 2019. SIF: A Framework for Solidity Contract Instrumentation and Analysis. In *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 466–473.

[38] Chao Peng and Ajitha Rajan. 2019. CLTestCheck: Measuring Test Effectiveness for GPU Kernels. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 315–331.

[39] Chao Peng and Ajitha Rajan. 2020. Automated test generation for OpenCL kernels using fuzzing and constraint solving. In *Proceedings of the 13th Annual Workshop on General Purpose Processing using Graphics Processing Unit*. 61–70.

[40] Van-Thuan Pham, Marcel Böhme, Andrew Edward Santosa, Alexandru Razvan Caciulescu, and Abhik Roychoudhury. 2019. Smart greybox fuzzing. *IEEE Transactions on Software Engineering* (2019).

[41] sc forks. [n.d.]. sc-forks/solidity-coverage. https://github.com/sc-forks/solidity-coverage, Accessed on: August 24, 2020.

[42] David Siegel. [n.d.]. Understanding the DAO attack. http://www.coindesk.com/understanding-dao-hack-journalists,Accessed on: April 19, 2020.

[43] SmartBugs. [n.d.]. SmartBugs: A Framework to Analyze Solidity Smart Contracts. https://smartbugs.github.io/,Accessed on: August 24, 2020.

[44] Nick Szabo. 1997. The idea of smart contracts. *Nick Szabo's Papers and Concise Tutorials* 6 (1997).

[45] Darrell Whitley. 1994. A genetic algorithm tutorial. *Statistics and computing* 4, 2 (1994), 65–85.

[46] Gavin Wood. 2014. A secure decentralised generalised transaction ledger [J]. *Ethereum project yellow paper* 151 (2014), 1–32.

[47] Michal Zalewski. 2017. American fuzzy lop (AFL). *URL: http://lcamtuf. coredump.cx/afl* (2017).